Tutoriat SO 8: Scheduling

December 11, 2025

Contents

1 Multiprocessor scheduling

In a multiprocessor context threads may actually run in parallel on a different core/cpu. We need to make sure that the load is spread as equally as possible to all cores/cpus.

Examples of multiprocessor systems:

- Multicore CPUs
- Multithreaded cores
- NUMAsystems
- Heterogeneous multiprocessing

1.1 Approaches

We can asymmetric scheduling, where one cpu only runs kernel code and the rest run user code. But this can create bottlenecks if for example all user cpus make a syscall at the same time.

In modern systems symmetric scheduling is used, where any thread can run on any cpu. Organizing the threads boils down to two approaches:

1. All threads may be in a common ready queue. 2. Each processor may have its own private queue of threads.

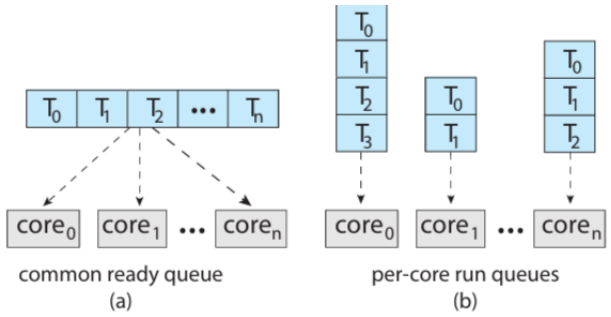

Figure 1: Thread organization multiprocessor systems

The first approach can create a race condition if two cpus(or better said, two instances of the scheduler running on 2 cpus) try to schedule the same thread at the same time. Can problem can be solved with locking but that would create a performance bottleneck.

The second approach(used in modern systems) solves this problem and even gives improved performance because of how caching works. The downside of this approach is the need for load-balancing, which we will discuss later.

1.2 Multicore systems

In modern systems we have multiple cpus each having multiple cores. Each core has it's own architectural context(has it's own registers) and appears as a separate CPU to the operating systems. These cores share L2/l3 caches which means the number of cache misses increases, which means the waiting time increases. The time wasted waiting for memory to become available is called **memory stall**.

Note: These cache misses can't be handled by the operating systems as no syscall is really made.

//TODO: Discuss about what kind of stalls the operating system actually handles.

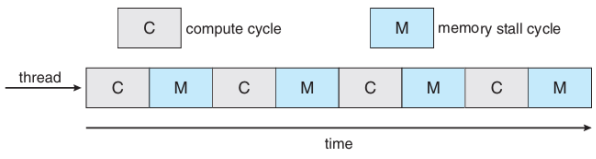

Figure 2: Memory stall of 50%

To reduce the impact of this, hardware threads were created, each core having multiple of these. If one of the threads stalls, execution can switch to another without going through the operating system.

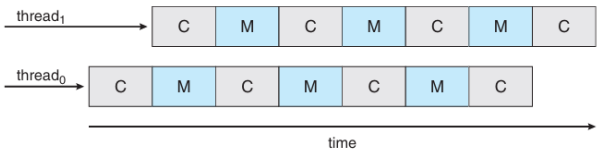

Figure 3: Hardware threads solution to memory stall

From the OS perspective, hardware threads also have their own execution context, which means each thread appears as a different cpu that is available to run a software thread. This technique is called **chip multi-threading (CMT)**

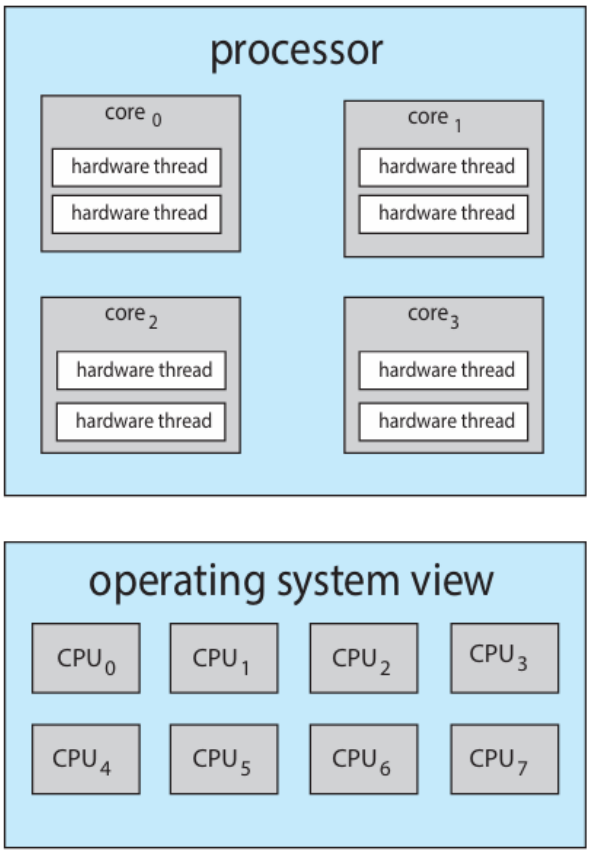

Figure 4: OS perspective of chip multi-threading

Note: Cores are the ones actually doing the "execution part", not cpus, not threads.

There are two types of multi-threading, **coarse-grained** and **fine-grained** multi-threading.

Coarse-grained: Thread executes on a core until a long-latency event occurs(such as a memory stall). When a stall is detected it's instruction pipeline is flushed. Another thread is chosen for execution and the pipeline is refilled. This operation is costly.

Fine-grained: Every few cycles a different thread is chosen to run on a core (this is done at the hardware level). This avoids the costly flush and refill pipeline operation from earlier but requires special hardware for doing this kind of thread switching.

Note: If you don't know what a pipeline is, it was presented at ASC in the first year

As pipelines and caches are shared among the threads on a core we have to do two level scheduling.

First level scheduling is done at the OS level with algorithms presented last time.

Second level scheduling happens at the core level, each core choosing which hardware thred should run.

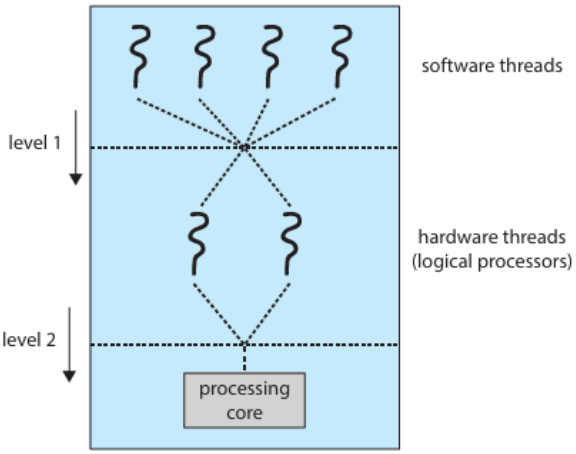

Figure 5: Multilevel scheduling example

1.3 Load balancing

Load balancing tries make sure that each logical cpu is used equally.

We have two approaches, **push migration** and **pull migration**.

Push migration: A task periodically checks the state of each cpu and pushes tasks from the busier to more idle ones.

Pull migration occurs when an idle processor pulls a waiting task from a busy processor.

// TODO DISCUSS DIFFERENT METRICS FOR LOAD

1.4 Processor affinity

If a thread runs on a specific processor it's cache gets populated with the data the threads needs. Which means memory accesses will be fast. If the thread gets moved to another processor, the old cache needs to be invalidated and the new one re-populated. This is a costly operation that needs to be avoided. This technique of assigning a thread to a specific processor is called processor affinity and is supported on most modern systems.

There 2 types of processor affinity:

1. Soft: The system tries it best to keep the thread on the same cpu but does not guarantee it.

2. Hard: The system makes sure a given thread only runs on a given list of processors.

1.5 Heterogeneous multiprocessing

This just means cpus can have cores that have different power consumptions. Cores with lower consumption can be used for threads that need to run a lot but don't do intensive operation (like daemons, background processes). Cores with higher consumption can be used for short periods of intensive operations(perhaps matrix multiplication, though a GPU would do a better job).

2 Real-Time Scheduling

There are two types of real-time operating systems (RTOS): **hard** and **soft**.

Hard RTOS must ensure that tasks are completed within a given time frame (deadline). If a task misses its deadline, the service is considered useless.

Soft RTOS do not guarantee strict timing, but they ensure that critical tasks receive service before non-critical ones (preemptiveness).

Next, we discuss some of the challenges that scheduling presents in RTOS.

2.1 Minimizing Latency

Because of the event-driven nature of RTOS, one problem that arises is **event latency**, the time between the occurrence of an event and the start of its handling.

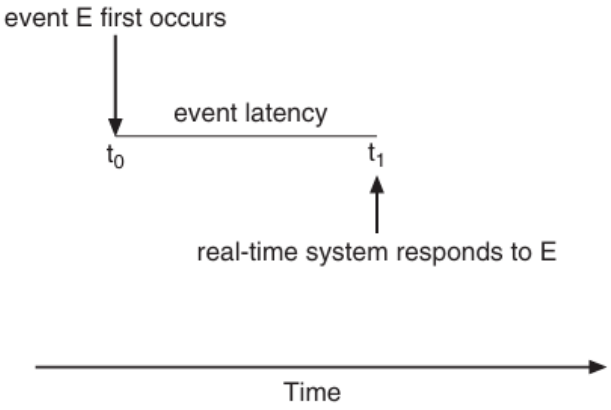

Figure 6: Event latency

There are two types of latencies: **interrupt latency** and **dispatch latency**.

2.1.1 Interrupt Latency

Interrupt latency is the time between the moment an interrupt occurs (whether software—soft IRQ—or hardware—hard IRQ) and the moment the corresponding interrupt service routine (ISR) begins execution.

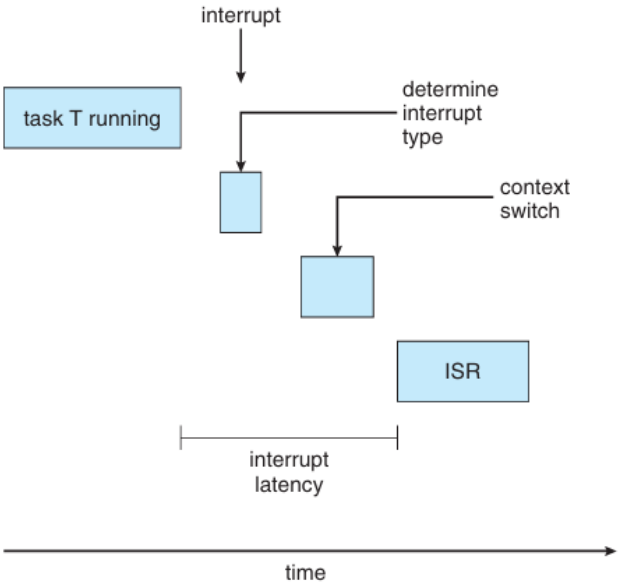

Figure 7: Interrupt latency

Question: How can we minimize this latency?

2.1.2 Dispatch Latency

Dispatch latency is the time required for the scheduler to stop one process and start another.

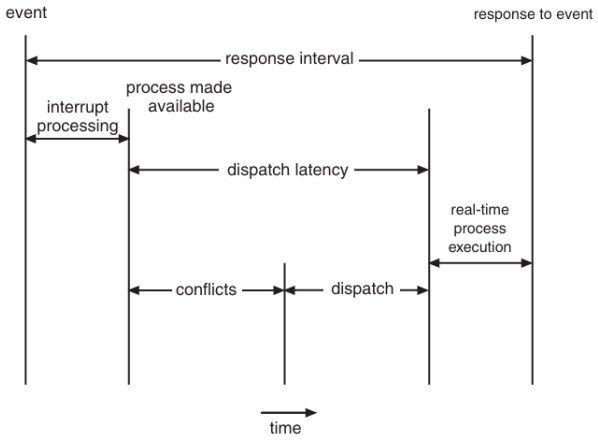

Figure 8: Dispatch latency

The conflict phase consists of:

1. Preempting any process running in the kernel.
2. Low-priority processes releasing resources required by a high-priority process.

Question: How can we minimize this latency?

2.2 Scheduling Algorithms

The most important feature of a real-time operating system is its ability to respond immediately to a real-time process as soon as the process requires CPU time. For this purpose, priority-based algorithms are used. These algorithms generally make the system soft real-time but not necessarily hard real-time.

In this context, processes have a fixed period p, burst time t, and deadline d. The inverse of the period $\frac{1}{p}$ is called the **rate**, indicating how often the process executes. Their relationship satisfies:

$$0 \leq t \leq d \leq p.$$

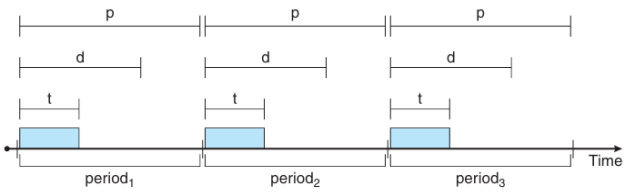

Figure 9: Periodic process execution example

2.2.1 Rate Monotonic Scheduling (RMS)

This strategy assigns priorities based on execution rates: processes with higher rates receive higher priorities.

Consider two processes P_1 and P_2 with:

$$t_1 = 20,\ p_1 = 50, \qquad t_2 = 35,\ p_2 = 100.$$

Deadlines occur at the start of each new period. The CPU utilization is:

$$\frac{t_1}{p_1} + \frac{t_2}{p_2} = \frac{20}{50} + \frac{35}{100} = 0.75.$$

If we incorrectly assign a higher priority to P_2, then P_1 misses its first deadline at time 50 because P_2 executes from 0 to 35 and P_1 only from 35 to 56.

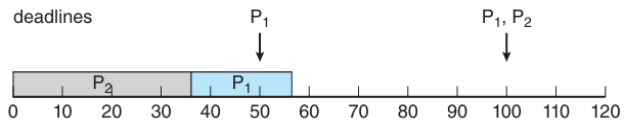

Figure 10: Non-monotonic scheduling example

Using RMS, P_1 has higher priority. It completes its first burst at time 20 and meets its first deadline. Then P_2 runs from time 20 to 50, is preempted by P_1, and resumes later, meeting its deadline.

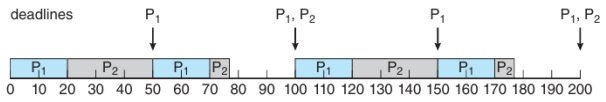

Figure 11: Monotonic scheduling example

RMS is optimal among all static-priority algorithms but does not guarantee that all processes meet deadlines if CPU utilization exceeds a certain limit.

The CPU utilization upper bound for n processes is:

$$n\left(2^{1/n}-1\right).$$

For 1 process the bound is 100%, for 2 it is 83%, and as $n \to \infty$ it converges to approximately 69%.

Failure Example Let $t_1 = 25$ and $p_2 = 80$. The CPU utilization becomes:

$$\frac{25}{50}+\frac{35}{80}=0.94.$$

This exceeds the RMS bound, and indeed P_2 misses its deadline at time 80.

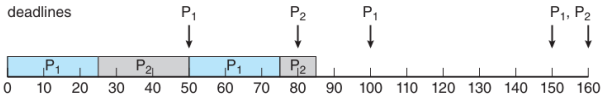

Figure 12: Rate monotonic scheduler failure

2.2.2 Earliest Deadline First Scheduling (EDF)

This algorithm does not require constant burst times nor process periodicity. It only requires that processes announce their next deadline when they become runnable. As the name suggests, priorities are assigned based on deadlines: the earlier the deadline, the higher the priority.

Let us reconsider the example for which rate-monotonic scheduling failed.

- Process P_1 has the earliest deadline, so its initial priority is higher than that of process P_2.
- Process P_2 begins running at the end of the CPU burst of P_1.

However, unlike rate-monotonic scheduling (which would allow P_1 to preempt P_2 at time 50), EDF scheduling allows P_2 to continue running. At this moment, P_2 has a higher priority because its next deadline at time 80 is earlier than the deadline of P_1 at time 100. Thus, both P_1 and P_2 meet their first deadlines.

Process P_1 resumes execution at time 60 and completes its second CPU burst at time 85, meeting its second deadline at time 100. Process P_2 runs afterward but is preempted

at time 100 when P_1 starts its next period. This preemption occurs because P_1 now has an earlier deadline (time 150) compared to P_2 (time 160).

At time 125, P_1 completes its CPU burst, allowing P_2 to resume. P_2 finishes at time 145, meeting its own deadline. The system remains idle until time 150, when P_1 is scheduled to run again.

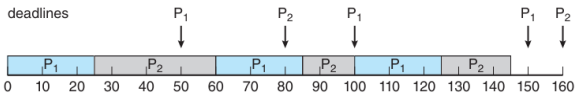

Figure 13: Earliest deadline first (EDF) example